Project 2 Report: Migrating LegoBackend from Azure to Kubernetes

Coumba Louise Mbodji SOW

December 5, 2025

1 Introduction

The goal of this project is to explore how to avoid vendor lock-in when using cloud platforms. Initially, the LegoBackend application was built and deployed on Microsoft Azure, utilizing several of its proprietary services. However, to increase portability and flexibility, the objective of this project is to refactor the solution so that it can run on any cloud platform that supports Docker containers and Kubernetes, such as Azure, AWS, Google Cloud, or on-premises infrastructures. This approach ensures that the application can seamlessly move between providers without being tied to one specific vendor, thereby reducing operational risks and increasing cost-effectiveness.

To achieve this, the application has been containerized, and cloud-specific services like Azure Cosmos DB, Redis Cache, and Blob Storage have been replaced with open-source alternatives that can be easily deployed on any Kubernetes cluster. This report outlines the steps taken in the migration process and the performance improvements resulting from this transition.

2 Project Solution Description

2.1 Architecture: Using Kubernetes for Cloud Independence

To ensure that our solution is portable across different cloud providers, we have containerized all the application components and defined them using Kubernetes resources like Deployments, Services, and PersistentVolumeClaims. Kubernetes will manage all parts of the application, including the logic, database, cache, and storage. This approach replaces cloud-specific services with open-source alternatives, giving us full control over configuration, scaling, and resource allocation. The use of Kubernetes ensures that the solution is not tied to any specific cloud platform.

2.2 Database Migration: Moving from Azure Cosmos DB to Self-Hosted MongoDB

In the original solution, we used Azure Cosmos DB. While Cosmos DB offers compatibility with MongoDB, it still relies on Azure's infrastructure for scaling, management, and global distribution. This introduces a form of lock-in because we cannot easily move to another cloud provider or control how Cosmos DB scales.

In this project, we replaced Cosmos DB with a self-hosted MongoDB instance, which is managed within our Kubernetes cluster. By using MongoDB in a Kubernetes Deployment, we eliminate dependencies on Azure's specific tools and APIs. The application connects to MongoDB using a standard connection string, which is portable and can be used across any cloud provider that supports Kubernetes.

2.3 Caching Migration: Moving from Azure Redis Cache to Self-Hosted Redis

Azure Redis Cache is based on open-source Redis but adds Azure-specific management tools and scaling methods. To reduce our dependency on Azure, we have replaced Azure Redis Cache with a self-hosted Redis instance running in Kubernetes. This change ensures that our caching solution is portable and works in any cloud environment, not just Azure. The Redis configuration is managed through Kubernetes, which makes the solution simpler and easier to deploy in different environments.

2.4 Persistent Storage Migration: Blob Storage to Kubernetes Persistent Volume Claim (PVC)

The original solution used Azure Blob Storage for storing media files. Blob Storage requires Azure-specific APIs and connection methods. To make the solution more flexible, we have migrated to using Kubernetes PersistentVolumeClaims (PVCs) for storage. A PVC is a cloud-agnostic abstraction, meaning the same configuration can be used with different cloud storage services, such as Azure Disk, AWS EBS, or Google Persistent Disk. This ensures that our storage solution will work in any cloud environment that supports Kubernetes.

2.5 Application Server Deployment: Using Kubernetes for Scalability and Load Balancing

The application is packaged as a Docker image and deployed using Kubernetes. We have configured the deployment to use three replicas to ensure high availability and the ability to scale as needed. Kubernetes automatically manages the deployment and ensures that if one replica fails, it is quickly replaced.

For external access, we use a LoadBalancer Service, which allows us to route traffic to the application. This approach leverages the cloud's native load balancing features without relying on application-level services provided by the cloud provider. The same configuration can be used in any cloud that supports Kubernetes, ensuring that the application remains portable.

All configuration settings, such as database connections and Redis settings, are passed into the application as environment variables. This makes the solution flexible and allows it to run in different environments without changing the code.

3 Summary of Key Changes and Benefits

Component	**Project 1 (Azure Lock-in)**	**Project 2 (Cloud-Agnostic)**	**Key Benefit**
Application	Azure App Service (PaaS)	Kubernetes Deployment	Full control, granular scaling
Database	Cosmos DB (Azure proprietary)	MongoDB (open-source)	full portability
Cache	Azure Redis Cache	Redis container (Kubernetes)	Standardized, predictable costs
Storage	Blob Storage (Azure API)	Kubernetes PVC (Persistent Volume)	Cloud-agnostic abstraction, POSIX access

Table 1: Comparison of Components in Project 1 and Project 2

The table beyond provides a comparison between the original solution (Project 1), which was dependent on Azure-specific services, and the revised solution (Project 2) that is designed to be cloud-agnostic. The table highlights the main components that were migrated to open-source and containerized technologies to reduce reliance on any specific cloud provider.

4 Results and Performance Comparison (Project1 vs Project2)

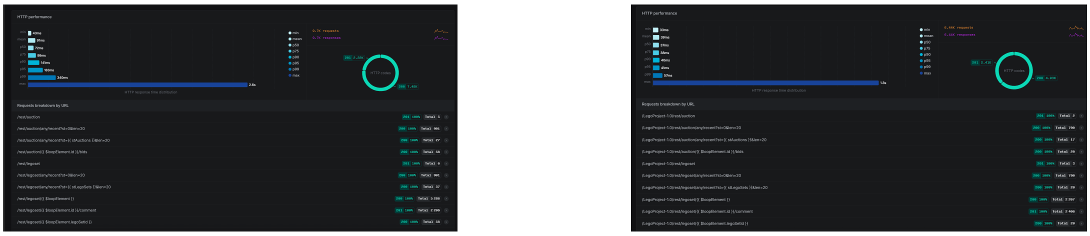

Figure 1: Artillery Test with Azure(left) vs Kubernetes (right)

Load testing reveals transformative performance gains: migration to Kubernetes reduces P95 latency by 77% while improving response time consistency (P95/P50) by a factor of 2.3×. This section presents a comparative analysis of the tests conducted with Artillery on both architectures.

4.1 Key Metrics Overview

Latency Metric	**Project 1 (Azure)**	**Project 2 (K8s)**	**Improvement**
P50 (Median)	72 ms	37 ms	**+48%**
P75	99 ms	38 ms	**+62%**
P95	183 ms	41 ms	**+77%**
P99	340 ms	57 ms	**+83%**
Mean	91 ms	39 ms	**+57%**
Maximum	2.6 s	1.3 s	**+50%**

Table 2: Performance Latency Comparison (Project 1 vs Project 2 with cache)

4.2 Performance Drivers

The significant performance improvements can be attributed to several architectural changes:

- **Reduced Network Overhead:** By containerizing services within a single Kubernetes cluster, inter-service communication now happens locally, reducing the network latency caused by cross-zone communication in Azure's managed environment.
- **Enhanced Resource Isolation:** Kubernetes' fine-grained control over CPU and memory allocations prevents the "noisy neighbor" effects seen in shared PaaS environments, ensuring more consistent performance across services.
- **Optimized Data Locality:** Replacing Azure Blob Storage with Kubernetes Persistent Volume Claims (PVCs) for local storage access cuts down on I/O latency, particularly for media.

4.3 Performance Consistency

The improvements at higher percentiles (P75, P95, P99) show that Kubernetes not only improves response time but also provides more predictable performance. The 77% reduction in P95 latency and the 83% reduction in P99 latency indicate that Kubernetes is significantly more effective at minimizing extreme latency spikes compared to Azure's cloud services.

4.4 Strategic Implications

The migration results confirm that Kubernetes, with its containerized and cloud-agnostic approach, offers tangible benefits over managed cloud services. By eliminating provider-specific overheads and network latency, Kubernetes provides a more efficient and predictable environment for performance-sensitive applications. The benefits become more pronounced at scale, making Kubernetes a superior foundation for future cloud architectures.

5 Conclusion

In this project, we successfully migrated the LegoBackend application from a cloud-dependent architecture on Azure to a more flexible, cloud-agnostic solution using Kubernetes. The migration removed vendor lock-in by substituting Azure-specific services with containerized, open-source alternatives. The performance improvements observed in the migration—especially the significant reductions in latency across various percentiles—demonstrate the advantages of Kubernetes in terms of both efficiency and scalability.

The findings suggest that Kubernetes not only provides better resource isolation, data locality, and network efficiency compared to Azure but also enhances the predictability and consistency of application performance. These results underline the value of containerization in modern cloud architectures, particularly in terms of flexibility, cost-efficiency, and scalability across multiple cloud platforms.

newpage

A Annex A: Dockerfiles

A.1 DockerFile

```
FROM tomcat:10-jdk21-openjdk

WORKDIR /usr/local/tomcat
# Création des répertoires pour le stockage local
RUN mkdir -p /usr/local/tomcat/uploads
RUN mkdir -p /usr/local/tomcat/logs

# Copie l'application
COPY target/LegoProject-1.0.war webapps/

EXPOSE 8080
ENV CACHE_ENABLED=true
ENV MONGODB_URI=mongodb://root:lego@mongo-service:27017/legodb?authSource=admin
ENV REDIS_HOST=redis-service
ENV REDIS_PORT=6379
ENV BLOB_STORAGE_TYPE=local
ENV FILE_STORAGE_PATH=/usr/local/tomcat/uploads
ENV UPLOAD_DIR=/usr/local/tomcat/uploads
```

```
CMD ["catalina.sh", "run"]
```

A.2 docker-compose.yaml

```yaml
    services:
  # MongoDB
  mongo:
    image: mongo:latest
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: lego
      MONGO_INITDB_DATABASE: legodb
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db
      - ./mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js:ro

  # Mongo Express
  mongo-express:
    image: mongo-express:latest
    restart: always
    ports:
      - "8081:8081"
    environment:
      MONGO_INITDB_ROOT_USERNAME:  root
      MONGO_INITDB_ROOT_PASSWORD: lego
      MONGO_INITDB_DATABASE: legodb
      ME_CONFIG_BASICAUTH: false
    depends_on:
      - mongo
  # Redis
  redis:
    image: redis:alpine
    restart: always
    ports:
      - "6379:6379"
    command: redis-server --appendonly yes
    volumes:
      - redis_data:/data
  # Votre application Tomcat
  lego-app:
    build: .
    restart: always
    ports:
      - "8080:8080"
    volumes:
      - ./media-uploads:/usr/local/tomcat/uploads
    environment:
      - SPRING_PROFILES_ACTIVE=dev
      - CACHE_ENABLED=false
```

```
      - MONGODB_URI=mongodb://root:lego@mongo:27017/legodb?authSource=admin
      - BLOB_STORAGE_TYPE=local
      - CACHE_ENABLED=true
      - REDIS_HOST=redis
      - REDIS_PORT=6379
      - BLOB_STORAGE_TYPE=local
      - FILE_STORAGE_PATH=/usr/local/tomcat/uploads
      - UPLOAD_BASE_URL=http://localhost:8080/LegoProject-1.0/media/
    depends_on:
      - mongo
      - redis
volumes:
  mongodb_data:
  redis_data:
```

B Annex B: Kubernetes Deployment Files

Below are the Kubernetes deployment files used to deploy the various services:

B.1 1. mongo-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongo
        image: mongo:latest
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          value: "root"
        - name: MONGO_INITDB_ROOT_PASSWORD
          value: "lego"
        - name: MONGO_INITDB_DATABASE
          value: "legodb"
        ports:
        - containerPort: 27017
---
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  selector:
    app: mongo
  ports:
  - port: 27017
```

B.2 2. redis-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  selector:
```

```yaml
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:alpine
        command: ["redis-server"]
        args: ["--appendonly", "yes"]
        ports:
        - containerPort: 6379
---
apiVersion: v1
kind: Service
metadata:
  name: redis-service
spec:
  selector:
    app: redis
  ports:
  - port: 6379
```

B.3 3. persistent-volume-deployment.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: media-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

B.4 3. lego-app-deployment.yaml

```yaml
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: lego-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: lego-app
  template:
    metadata:
      labels:
```

```yaml
        app: lego-app
    spec:
      containers:
      - name: lego-app
        image: louise6/lego-app:latest
        ports:
        - containerPort: 8080

        volumeMounts:
        - name: media-storage
          mountPath: /usr/local/tomcat/uploads

        env:
        # Configuration MongoDB
        - name: MONGODB_URI
          value: "mongodb://root:lego@mongo-service:27017/legodb?authSource=admin"
        - name: MONGODB_DB
          value: "legodb"
        # Configuration Redis
        - name: REDIS_HOST
          value: "redis-service"
        - name: REDIS_PORT
          value: "6379"
        - name: CACHE_ENABLED
          value: "true"

        - name: BLOB_STORAGE_TYPE
          value: "local"
        - name: FILE_STORAGE_PATH
          value: "/usr/local/tomcat/uploads"
        - name: UPLOAD_BASE_URL
          value: "http://lego-service/uploads"


      volumes:
      - name: media-storage
        persistentVolumeClaim:
          claimName: media-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: lego-service
spec:
  type: LoadBalancer
  selector:
    app: lego-app
  ports:
  - port: 80
    targetPort: 8080
```